Extending a Deductive Object-Oriented Database System with Spatial Data Handling Facilities

Alvaro A.A. Fernandes
Department of Computer Science
University of Manchester

Andrew Dinn
Department of Computing and Electrical Engineering
Heriot-Watt University

Norman W. Paton
Department of Computer Science
University of Manchester

M. Howard Williams
Department of Computing and Electrical Engineering
Heriot-Watt University

Olive Liew
Department of Computing and Electrical Engineering
Heriot-Watt University

December 2, 1998

Abstract

This paper describes the integration of a spatial data handling component with the ROCK & ROLL deductive object-oriented database system. The extended ROCK & ROLL system provides much more comprehensive and better integrated database programming facilities than other candidate platforms for spatial information systems. The extended system serves developers with an intuitive, expressive, formally defined collection of spatial data types as primitive types whose operations have state-of-the-art computational complexity. The integration of these types with the object-oriented modelling, imperative programming and deductive querying facilities of ROCK & ROLL makes available a comprehensive and integrated suite of complementary mechanisms for the development of spatial information systems. The paper also provides preliminary benchmark results which indicate that kernel-support for spatial data handling does yield performance gains and that the extended ROCK & ROLL system compares well with a specialist geographic information system and two widely-known extensible database systems when the latter are extended with spatial data handling facilities.

Keywords: *Spatial Data Management, Realms, Spatial Data Types, ROSE Algebras, Deductive Object-Oriented Databases, Database Programming Languages, ROCK & ROLL*

1 Introduction

Spatial data handling has long posed challenges to database technology. In particular, the rapidly growing use of spatial (or geographic) information systems (SISs) (i.e., information systems that depend crucially on storing, querying and manipulating spatially-referenced data) has helped expose certain shortcomings of contemporary, mainstream database technology. The challenges presented by SISs stem from both the structural complexity of the data that must be stored and the algorithmic complexity of the operations on that data. Contemporary, mainstream database management systems (DBMSs) fall short of what is needed due both to spartan underlying data models which do not adequately cope with the structural complexity of spatial data and to query and manipulation languages which are not expressive or user-friendly enough to operate smoothly on algorithmically complex spatial operations.

Most responses to these shortcomings have followed a coupling approach in which a distinct, independent, loosely-coupled spatial data handling component is bolted on to a DBMS (e.g., [27]). This lack of integration has often led to poor productivity in the development and use of SIS applications and to poor runtime performance. Productivity is hindered because of the need imposed on programmers and end-users to mediate between independent components. Runtime performance is compromised because the DBMS cannot make intelligent use of the semantic properties of the spatial data types (SDTs) supported by a loosely-coupled spatial data handling component. To see why, consider the fact that an algebra determines equivalences between some of its expressions. These equivalences are of great interest to an optimizer since two semantically equivalent queries may have very contrasting work loads associated with them. By definition, in a loosely coupled approach, the optimizer has no knowledge of the spatial algebra and the spatial data handling component cannot easily take responsibility for exploiting semantic equivalences because it requires statistical information that is only available within the DBMS. To remedy this, one might adopt an approach in which the DBMS and the spatial data handling component exchange information for the purposes of optimization. However, the communication overheads associated with this approach cannot but increase the cost of optimizing spatial queries thereby reducing the class of such queries that can be efficiently optimized. This leads to opportunities for optimization being missed, thereby compromising the overall quality of support that the system provides.

In order to overcome the problems associated with a coupling approach it is necessary to devise ways of integrating a spatial data handling component with a host DBMS as seamlessly as possible. Choices that arise at this point include:

Which host DBMS? Its standard components (in particular, the storage manager, the query optimizer and the query evaluator) must be sufficiently expressive, flexible and extendable.

Which SDTs? These should be expressive and make up a collection whose application to SIS development is intuitive and grounded on a simple, formal semantics. At the same time, the operations on spatial objects should have acceptable computational complexity.

Ultimately, these choices have to be validated by a demonstration that the chosen host DBMS and the chosen SDTs are compatible, in the specific sense that the data model and the query and manipulation languages of the host DBMS are capable of accommodating the SDTs and operations without introducing impedances to the productivity of SIS development, while maximizing the opportunities for enhancing runtime performance.

This paper describes the integration of a spatial data handling component based on the ROSE (RObust Spatial Extensions) approach described in [19, 20, 21] with the ROCK & ROLL deductive object-oriented database (DOOD) system [7, 8, 12]. Therefore, the paper argues for the choice of ROCK & ROLL as a suitable host DBMS and for a ROSE algebra as providing a suitable collection of SDTs.

The ROSE approach to spatial data handling is comprehensively formalized within set theory as a 5-tiered algebra of spatial objects defined with the specific intention of facilitating its integration with general-purpose DBMSs, so as to yield spatial DBMSs with the following desirable characteristics:

- The DBMS into which the ROSE algebra is integrated loses none of its general-purpose database functionality. The incorporation of spatial data types is minimally intrusive, and requires no changes – only extensions – to the existing ROCK & ROLL model and languages.
- The spatial component of the extended DBMS is defined over a model of geometry that both addresses the limitation that computers can only represent numbers discretely and facilitates the design of efficient algorithms that implement operations on spatial data. These algorithms are available [19].

The ROCK & ROLL system implements a DOOD system using a persistent language approach. Architecturally, it consists of three components:

1. The OM persistent store, which implements the object-oriented data model underlying ROCK & ROLL as a set of persistent (C++) classes.
2. The ROCK interpreter, which implements the data-definition language (DDL) and the imperative data-manipulation language (DML) of the system.
3. The ROLL interpreter, which implements the deductive query language (DQL) of the system.

All the ROCK & ROLL components were implemented in the wake of a comprehensive presentation of their formal semantics [16]. ROCK & ROLL, including the extensions described in this paper, is publicly available through the Internet (`http://www.cee.hw.ac.uk/Databases/rnr.html`).

The main contribution of this paper is to show that the pair (ROCK & ROLL, ROSE) is an instance of a compatible (host-DBMS, SDT) pair in the sense previously defined and that their integration leads to an effective and efficient platform for the development of SISs. From the point of view of SIS developers and users the most important aspect of this integration task is that the syntactic and semantic treatment of an SDT and its instances is consistent with that of ROCK & ROLL types and their instances. In particular, instances of SDTs are treated as primitive values in ROCK & ROLL. The extended ROCK & ROLL system accords to spatial values a privileged status that is consistent with that accorded to booleans, integers, reals and strings.

In terms of research into DOODs, this paper shows how the ROCK & ROLL DOOD system is amenable to the incorporation of spatial data types, and that such extensions do not require complex or unintuitive extensions to its deductive language, its object-oriented model, or, indeed, the implementations of these components.

For reasons of space this paper cannot but assume some familiarity with [7, 8, 12] and with [19, 20, 21]. However, brief overviews of both the ROCK & ROLL DOOD system and the ROSE approach are given below which should suffice for the understanding of later sections. The remainder of this paper is structured as follows. Sections 2 and 3 provide brief overviews of ROCK & ROLL and the ROSE approach, respectively. Section 4 details the extensions to ROCK & ROLL and exemplifies their use by presenting a simple fire emergency application programmed in ROCK & ROLL. Section 5 discusses related work. Preliminary benchmark results are reported in Section 6. Finally, Section 7 draws some conclusions.

2 A Brief Overview of ROCK & ROLL

The most distinctive feature of ROCK & ROLL as a DOOD system is the fact that it implements an expressive object-oriented data model (known as OM) from which two distinct languages, one imperative (known as ROCK) and one deductive (known as ROLL), are derived. OM supports object identity, single and multiple structural and behavioural inheritance and bulk-type constructors (allowing the specification of set-, list- and tuple-like application types). ROCK supports typical control constructs (e.g., *foreach*, *while*, *if*) and standard facilities to effect state transitions, such as destructive assignment, input and output, and the explicit creation and deletion of application objects. ROCK is used for writing methods and free standing programs and applications. It provides all the necessary primitives to manipulate OM structures as built-in methods and operators. ROLL is a pure, function-free definite-clause language, equivalent to Datalog with stratification and range restriction to support negation and operations on primitive values [10].

Like Datalog, ROLL is order-independent and cannot effect state transitions. These characteristics maximize optimization opportunities and ensure that ROLL evaluation is tractable [5] and benefits from techniques adapted from a relational context into an object-oriented one [12]. Most of the differences between Datalog and ROLL arise from the object-oriented nature of OM, from which ROLL is derived, but these differences have been so designed as to preserve the desirable metalogical properties that ROLL shares with Datalog. ROLL is used for writing embedded queries, for defining implicit data and for writing side-effect free rule-based methods. Built-in methods and operators defined for ROCK which are side-effect free are also available in ROLL. Code in both languages is statically type-checked, and type inference is used to infer the type of the result of embedded ROLL queries. Methods written in either language persist in the database. Both languages support recursion and profit from standard behaviour sharing facilities such as overriding and dynamic binding. Since ROCK and ROLL are formally derived from OM the impedance mismatches that tend to affect multilanguage platforms do not manifest themselves in ROCK & ROLL [8].

Consider the ROCK & ROLL code fragments in Figure 1. They define a domain involving states (of a country) and their road network. The declaration of a type specifies the structure of, and a behavioural interface to, instances of the type. For example, in Figure 1(a) the type `state` has properties `name`, `network`, `counties`. The domain of a property appears to the right of the colon (which denotes type aliasing, i.e., the token to the left of the colon becomes a synonym for the type-denoting token to the right). Thus, the domain of `name` is the ROCK & ROLL primitive type `string`, that of `network` is the application type `roads` (defined immediately below), and that of `counties` is a spatial type `regions` (which, with `lines` and `points`, is assumed to come from a separately-implemented ROCK & ROLL library of SDTs, in the style of [1]). The application-type `roads` is a set-like bulk type constructed from the type `road` by *association* (a list-like type would have used square brackets and a tuple-like one angle brackets). Inheritance is supported as shown: the type `motorway` is a subtype of `road`. Like `state`, both `road` and `motorway` have spatial properties.

Notice in the declaration of `state` the keywords `ROCK` and `ROLL` which introduce the behavioural part of the type declaration. Instances of `state` respond to the message `roads_at_level` with a (new) instance of `roads` comprising all roads in the state at the level passed as a parameter in the invocation. Also, instances of `state` respond to the message `has_road`. In this case, however, because ROLL is a deductive language[1], the method can be invoked with different, alternative patterns of bound and unbound arguments. For example, associating to the type `state` the signature `has_road(road)` effectively means that one can use it to test whether a given road belongs to a given state, to find all states to which a given road belongs, to find all roads in a given state and to associate to each and every road with each and every state it belongs to. These queries correspond, respectively, to the following `road`—`state` binding patterns: bound-bound, bound-unbound, unbound-bound and unbound-unbound. Notice also that the signature of a ROCK method associates to the type of an argument, the local name by means of which the parameter can be referenced within the body of the method, whereas the signature of a ROLL method only declares the type of its arguments, because in its body only logical variables occur and there is no need for names local to the method.

Figure 1(b) shows an implementation for the interface of `state` and illustrates method writing in ROCK & ROLL.

[1] See [10] for a description of variable binding in deductive languages.

```
type state:
   properties:
      public:
         name     : string,
         network  : roads,
         counties : regions;
   ROCK:
      roads_at_level(l: string): roads;
   ROLL:
      has_road(road);
end-type # state

type roads:
   public
      { road };           # interface omitted
end-type # roads

type road:
   properties:
      public:
         name  : string,
         level : string,
         route : lines;   # interface omitted
end-type # road

type motorway:
   specialises: road;
   properties:
      public:
         cafes : points;  # interface omitted
end-type # motorway
```

(a) Type Declarations

```
class state
   public:
      roads_at_level(l: string): roads
      begin

         var scope := get_network@self;
         var result:= new roads();

         foreach r in scope
         do if   (get_level@r = l)
            then result ++ r;

         result

      end

      has_road(road)
      begin

         has_road(R)@S :-
            get_network@S == Network,
            R is_in Network;

      end
 end-class # state

 ...                # other classes omitted
```

(b) Class Declarations

Figure 1: Example Schema in ROCK & ROLL

Local variables, e.g., `scope`, can be declared and used. A built-in method, denoted by `new`, is used for explicit creation of objects. The control construct `foreach` ⟨*controlVar*⟩ `in` ⟨*collection*⟩ `do` ⟨*block*⟩ can be used to iterate over the objects in a class or over the components of an object, e.g., `scope`. A special denotation, viz. `self`, is provided for the recipient of the message being attended to. The language has destructive assignment (using the `:=` operator). Built-in operators are exemplified by the infix operator `++`, which adds to the set-like object on its left the object on its right, whereas `--` would have removed it. For each declared property p the compiler generates two methods with name `get_`p and `put_`p that allow retrieval and update, respectively, of the value assigned to p (e.g., `get_network`). The value of the last evaluated statement is the value returned to the caller as an answer to the method invocation (e.g., `result` in `roads_at_level`). ROLL methods are written using a notation reminiscent of Prolog as shown by the implementation of `has_road`. Notice, besides the compiler-generated `get_network` predicate, the `is_in`/2 predicate that is the OM-equivalent to set membership and the `==`/2 predicate denoting unification.

```
var Level : string;
read Level;
foreach S in state
do if (get_name@S = "Idaho")
   then begin
          var Rs := roads_at_level(Level)@S;
          foreach R in Rs
          do write get_name@R, nl; # nl for new line
          delete Rs;
        end
```

(a) Using ROCK Only

```
var Level : string;
read Level;
var R_Ns : { string };

R_Ns := [{R_N} | has_road(R:road)@S,
                 get_name@R == R_N,
                 get_level@R == !Level,
                 get_name@S == "Idaho"];
foreach R_N in R_Ns
do write R_N, nl;
```

(b) Using ROCK and ROLL

Figure 2: Data Retrieval in ROCK & ROLL

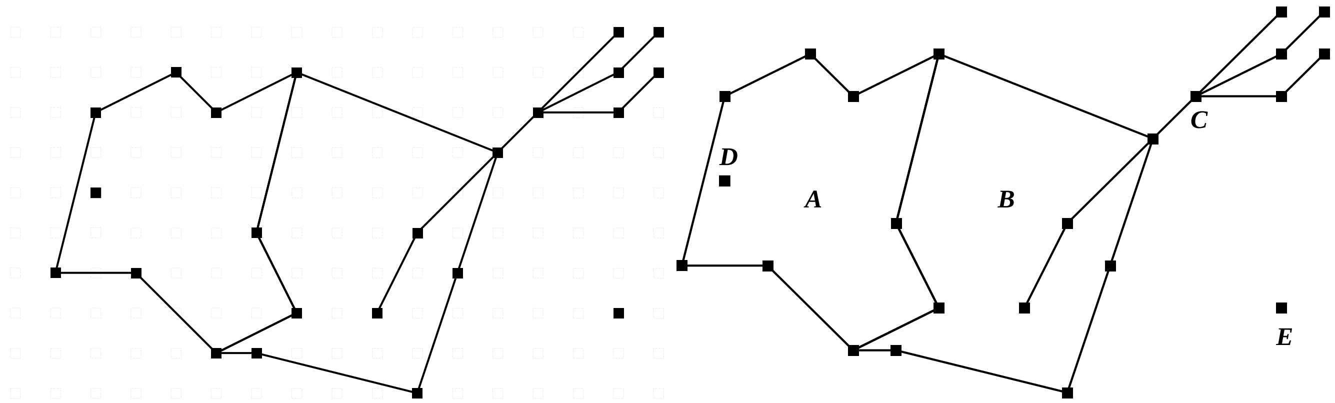
Figure 2 shows how ROCK & ROLL makes viable alternative styles for retrieving data in applications. The code fragments in Figure 2(a) and (b) are equivalent, in the sense that they produce the same results. Since (a) uses an imperative style to define explicitly how data is to be retrieved, query optimization does not take place. In contrast, (b) uses an embedded ROLL query which *is* optimized so as to maximize the chances of achieving good performance irrespective of the current state of the database. An embedded query has, in addition to the conjunction of atoms to the right of the vertical bar, a projection expression that specifies how the answer is handed over to the calling code, both in terms of variables and of the kind of collection, if any. Notice the absence of type impedance insofar as the result of the embedded query is stored in a variable, `R_N`, which the subsequent `foreach` construct can iterate over. The presence of curly brackets indicates that *all* solutions are wanted, their absence would have indicated that *any* solution would suffice and an empty projection indicates that only a *boolean response* is required. Notice also that a type annotation on a variable (e.g., `R:road`) acts as a constraint when type extensions have to be scanned during query evaluation. Notice, finally, that the values of ROCK state variables make their way into queries as bound ROLL logical variables by being prefixed with the character '`!`'. For example, in the embedded ROLL query in Figure 2(b), the `!` in `!Level` signifies to the ROLL evaluator that it should access the value of the ROCK variable `Level`) defined earlier. The example also shows how ROLL methods can be called from ROCK. ROCK methods can also be invoked from ROLL methods and queries provided that they do not effect any state transition at a global level, as explained in more detail in Section 4.

3 A Brief Overview of the ROSE Approach

The ROSE approach to spatial data modelling culminates in the definition of an algebra of SDTs. A ROSE algebra characterizes three kinds of spatial entities: *points*, *lines* and *regions*. The operations on each of these spatial entities are described in Section 4. Note that, as usual in database programming, a ROSE algebra models spatial entities as sets with a view to set-at-a-time processing. Thus, every spatial value in a ROSE algebra is a set value. Each element of a *points* value is a point defined by a pair of coordinates in the underlying geometry. Each element of a *lines* value is a set of connected line segments. Each element of a *regions* value is an areal object consisting of a boundary and a possibly empty set of inner regions, where both the boundary and the inner regions are represented by a cycle made of line segments.

The underlying geometric domain of a ROSE algebra is a finite discrete grid (built from integer values) called a *realm*. The spatial values manipulated by a ROSE algebra are defined over a realm. This approach requires the definition of a mapping of the Euclidean space presumed in most SIS applications into a realm. This mapping is well defined [20, 21] and has properties which guarantee that all spatial operations over realm values are error bound and only take, and yield, intersection-free spatial-values. This, in turn, means that ROSE-algebraic spatial operations have more efficient implementations than alternative approaches at the level of computational geometry [19]. Of course, developers may need to concern themselves with the mapping from non-integer data points to integer ones before the database is populated. Correspondingly, developers may want to map integer data points back to non-integer ones before displaying spatial objects.

Some examples of spatial objects taken from [20, 21] are given in Figure 3. Figure 3(a) is a graphical representation of an example realm. Figure 3(b) depicts some spatial objects defined over the realm in Figure 3(a), viz.: two polygonal objects **A** and **B**, one polyline object **C** and two point objects **D** and **E** defined by reference to the realm depicted in Figure 3(a). Thus, the following exemplify spatial values definable over the realm depicted in Figure 3: one *regions* value whose single element is **A**, another whose single element is **B**, the *lines* value that is the singleton containing **C** and the *points* value whose elements are **D** and **E**.



(a) A Realm State (b) Spatial Objects Defined Over a Realm State

Figure 3: Example of the ROSE Approach to Geometry

The piecemeal definition of a ROSE algebra consists of several layers with each successive layer relying on concepts defined in preceding layers. The topmost layer, i.e., the one that is exported to users, is called the *ROSE algebra* layer. The preceding layer, upon which it relies is called the *spatial abstract data type* (SADT) layer. In [21] the link between the SADT layer and the ROSE algebra defined over it is via an *object model interface* (OMI). This notion highlights the fact that the layers up to, and including, the SADT layer, are independent of any particular model of data and of any particular language. Coupling the SADTs defined over realms to a particular model of data and to a particular set of language constructs is what an OMI achieves. Therefore, the OMI must provide a set of data modelling concepts with which a ROSE algebra can be defined and a set of language constructs and notations that make a ROSE algebra concrete and, thereby, usable in practice.

Notice that, strictly speaking, what Güting and Schneider refer to in [21] as *the* ROSE algebra is more precisely, *a* ROSE algebra, insofar as different OMIs determine different ROSE algebras over the SADT layer. In fact, the OMI adopted in [21] is different from the one described in this paper in two respects:

- The data modelling concepts part of the OMI adopted in [21] is slightly slanted towards relational DBMSs, whereas the one described here is, like ROCK & ROLL, object-oriented.
- The language-constructs part of the OMI adopted in [21] is slightly slanted towards an SQL-like language, whereas the one described here is, like ROCK & ROLL, equipped with both imperative, state-based language constructs and declarative, deductive ones.

In the light of the remarks above, this paper can also be understood as describing an alternative OMI to that of [21], insofar as it shows how the same SADTs give rise to a different, ROCK & ROLL-embedded ROSE algebra.

All the concepts and notions referred to in this brief overview, plus all the spatial entities and operations defined in each layer, are given a comprehensive formal treatment in [20, 21].

4 Spatial Extensions to ROCK & ROLL

The most important decisions underlying the spatial extensions to ROCK & ROLL described in this paper were the following. Firstly, to integrate the support of spatial types at the level of the ROCK & ROLL kernel rather than to provide a spatial library that can be added as a layer between SIS applications and ROCK & ROLL. Finally, to treat instances of spatial types as primitive values rather than objects.[2]

The main advantages of integrating spatial types to the kernel, rather than layering them, are the following. Firstly, if spatial types are part of the kernel the DBMS has full knowledge of their semantics and can make the most of that knowledge to optimize storage strategies and query and method execution plans. Secondly, SIS developers can start from a much higher-level view of spatial entities than if these had to be implemented as a layer of ROCK & ROLL. Correspondingly, the main disadvantage of not layering spatial types is that SIS developers cannot easily adapt the treatment of geometry to specific circumstances.

The main advantage of treating spatial entities as values, as opposed to objects, is that the semantics of spatial entities and of spatial operations is completely known to the system. On the basis of this knowledge, the system can be designed to make use of methods and techniques to manage more efficiently both the execution of spatial operations and the automatic allocation and deallocation of storage space for instances of spatial types, thereby relieving SIS developers from such complex and error-prone tasks. The main disadvantage is the greater complexity of implementation and the potential for occasional inefficiency incurred in most generic optimization strategies and in implicit storage management strategies.

The following constraints hold for values of any primitive type in ROCK & ROLL, and hence also for instances of the ROSE SDTs in the extended system:

- SDTs can neither be refined nor redefined within the system's DDL.
- Instances of SDTs persist only if they occur in an instance of a user-defined persistent class.
- Instances of SDTs are neither created nor destroyed using ROCK & ROLL constructs explicitly.
- It is not possible to iterate over the extension of an SDT in ROCK & ROLL.

None of the above constraints hold for proper objects, i.e., instances of the specific types defined explicitly for an application which may or may not refer to instances of SDTs in their state. In other words, application-specific types can be refined and redefined, will persist if desired, can be explicitly created and destroyed and allow their extension to be iterated over, independently of whether or not they are spatially referenced via instances of SDTs.

The remainder of this section details how the ROCK & ROLL system is enriched with the ROSE SDTs. Using this strategy of treating the latter as primitive types in ROCK & ROLL, changes are needed to support new type declarations, spatial literals and operations on spatial values, while other aspects of the ROCK & ROLL languages remain essentially unchanged. The remainder of this section describes several such aspects and comments on the changes needed or not, as the case may be.

Type Declarations The original primitive types of ROCK & ROLL are *boolean*, *integer*, *string* and *real*. To support the ROSE SDTs, the set of reserved words of ROCK & ROLL is enlarged with `points`, `lines`, and `regions` to denote the ROSE SDTs *points*, *lines* and *regions*, respectively. At the semantic level, the ROCK & ROLL implementation of the OM data model is extended to support a 5-tiered ROSE algebra which follows very closely the one described in [19, 20, 21] with the improvements reported in [28].

The simplifying effect of this strategy is illustrated in Figures 1 and 2. From the point of view of an SIS developer, the extended ROCK & ROLL system supports the ROSE SDTs and spatial operations upon them in a manner that is consistent with the current support for booleans, integers, reals and strings. In contrast, in a loosely-coupled system, it falls on the shoulders of application programmers to consider the effects of impedance mismatches and to work around them. This makes applications more complex to design, develop, test and maintain.

An application object can have zero (e.g., `roads`) or more (e.g., `road`, `state`) spatial properties. An application object can also be a composite object made of spatially-referenced objects (e.g., `roads`). Interface declarations can involve spatially-referenced types in the expected way (e.g., `has_road(road)`). Like other primitive types in ROCK & ROLL the ROSE SDTs are operated upon by spatial operations that are provided to developers as built-in methods and operators. These are presented later in this section.

State Variables and State Transition There is no change in the way one declares a variable to store spatially-referenced objects (e.g., `R_Ns` in Figure 2(b)), and existing ROCK & ROLL syntax is used to input and output a spatial value to and from a variable. The `read` operation accepts from the input stream a textual representation

[2] Roughly, values denote application independent entities, while (proper) objects denote application dependent ones. The most general implication of this distinction is the application designer's having to declare and define explicitly the structure and behaviour of objects, but not of values, which have, in this sense, a given, fixed, predefined, well-defined structural and behavioural definition inbuilt into the programming language's own implementation.

of the spatial value and stores its internal representation. Conversely, the `write` operation converts the internal representation of a spatial value stored in a variable into its corresponding textual representation placing the latter in the output stream.

Assuming the declarations in Figure 1, Figure 4 shows two free-standing ROCK & ROLL programs that, to the left, populates the class `motorway` in the persistent environment `DB` and, to the right, iterates over its extension displaying each of its instances. The code fragments also illustrate some ROCK & ROLL built-in methods to handle external files (e.g., `openr`, `eof()`, `closer`).

```
program populate_motorways(DB)                          | program display_motorways(DB)
begin                                                   | begin
   openr("motorway.dat");                               |   foreach M in motorway
   while (not eof())                                    |   do begin
   do begin                                             |         write "motorway ", get_name@M,  nl;
         var N : string; # a name                       |         write "-------- ", nl, nl;
         var L : string; # a level                      |         write "    name ", get_name@M,  nl;
         var R : route;  # route is an alias for lines  |         write "   level ", get_level@M, nl;
         var C : cafes;  # cafes is an alias for points |         write "   route ", get_route@M, nl;
         var M : motorway;                              |         write "   cafes ", get_cafes@M, nl;
                                                        |      end
         read N;                                        | end
         read L;                                        |
         read R;                                        |
         read C;                                        |
         M := new motorway(N,L,R,C);                    |
         write "created motorway ", get_name@M, nl;     |
      end                                               |
   closer;                                              |
end                                                     |
```

Figure 4: Input and Output of Spatial Values in ROCK & ROLL

Spatial Literals In external files as in source texts, spatial values are representable by spatial literals. A `points` literal is an expression of the form

$$\texttt{points}\{p_1, \ldots, p_n\}$$

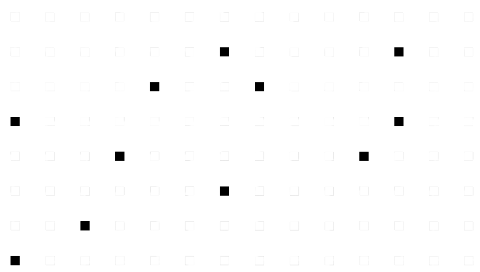
where each $p_i, 0 \leq i \leq n$, is represented as a pair of integer literals separated by a colon. Examples of `points` literals are given in Figure 5(a).

```
points{}

points{0:0}

points{
   0:0,  0:4,  2:1,  3:3,
   4:5,  6:2,  6:6,  7:5,
  10:3, 11:4, 11:6
}
```

(a)



(b)

Figure 5: `points` Spatial Literals

The first literal in Figure 5(a) denotes the empty `points` value. The second denotes a singleton `points` value. The last is depicted in Figure 5(b).

A `lines` literal is an expression of the form

$$\texttt{lines}\{b_1, \ldots, b_m\}$$

where each $b_i, 0 \leq i \leq m$ is a connected graph, expressed in the form

$$[s_1, \ldots, s_n]$$

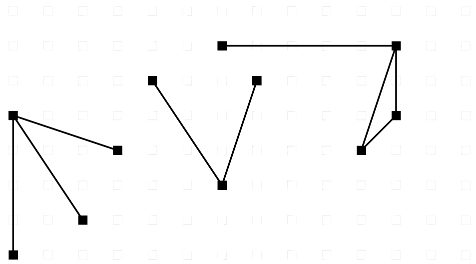
where each $s_i, 1 \leq i \leq n$, is a line segment represented as a pair of points (as previously described) separated by an underscore. Note that the syntax of literals does not imply any particular storage model. In practice, data points are stored once and thereafter simply referred to by application objects. Examples of `lines` literals are given in Figure 6(a).

```
lines{}

lines{[0:0_0:4]}

lines{
  [ 0:0_ 0:4,  2:1_ 0:4,  3:3_ 0:4            ],
  [ 6:2_ 4:5,  6:2_ 7:5                       ],
  [ 6:6_11:6, 11:6_11:4, 11:4_10:3, 10:3_11:6 ]
}
```

(a)



(b)

Figure 6: `lines` Spatial Literals

The first literal in Figure 6(a) denotes the empty `lines` value. The second denotes a singleton `lines` value. The last is depicted in Figure 6(b).

A `regions` literal is an expression of the form

$$\texttt{regions}\{f_1, \ldots, f_m\}$$

where each $f_i, 0 \leq i \leq m$, is represented as a pair of components separated by the keyword `inner`. The left-hand component represents the boundary of an area and the right-hand component is a set representing inner areas,[3] if any, within the boundary. Each left-hand component is an expression of the form

$$[s_1, \ldots, s_n]$$

where each $s_i, 1 \leq i \leq n$, is a line segment (as previously described). The right-hand component is an expression of the form

$$\{c_1, \ldots, c_p\}$$

where each $c_i, 0 \leq i \leq p$, has the same form as the left-hand component. If there are no inner areas then only the left-hand component needs to be written and the keyword `inner` is omitted. Examples of `regions` literals are given to the left of Figure 7.

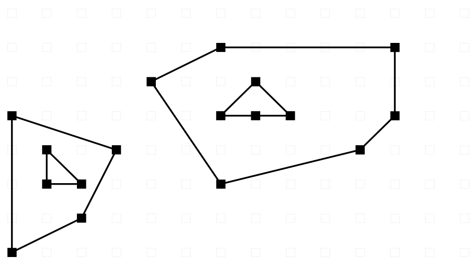
```
regions{}

regions{[ 0:0_2:1, 2:1_3:3, 3:3_0:4, 0:4_0:0 ]}

regions{
  [  0:0_2:1, 2:1_3:3, 3:3_0:4, 0:4_0:0 ]
  inner {[ 1:2_2:2, 2:2_1:3, 1:3_1:2 ]}
}

regions{
  [ 0:0_2:1, 2:1_3:3, 3:3_0:4, 0:4_0:0 ]
  inner {[ 1:2_2:2, 2:2_1:3, 1:3_1:2 ]},
  [ 6:2_ 4:5,  4:5_6:6, 6:6_11:6, 11:6_11:4,
   11:4_10:3, 10:3_6:2 ]
  inner {[ 6:4_7:4, 7:4_8:4, 8:4_7:5, 7:5_6:4 ]}
}
```

(a)



(b)

Figure 7: `regions` Spatial Literals

The first literal in Figure 7(a) denotes the empty `regions` value. The second denotes a singleton `regions` value, the only component in which has no inner regions inside (and hence only the left-hand part is written). The third

[3]The proponents of the ROSE approach talk metaphorically of these inner areas as *holes*. However, because the inner areas can themselves have inner areas, it is possibly better to imagine an inner area as a *lake in an island*, since a lake in an island may have islands which may have lakes, and so on.

denotes another singleton `regions` value, but this time there is a hole inside it (and hence the keyword `inner` and the right-hand component are also written). The last `regions` literal has two regions, each with a hole, and is depicted in Figure 7(b). Examples of statements involving literals are shown in Figures 8 and 9.

Spatial Operations The SADT layer of [20, 21] formally defines all the spatial operations that can be performed on instances of *points*, *lines* and *regions*. Algorithms in Modula-2 that implement them are given in [19]. All the spatial operations defined in the SADT layer of [20, 21] are supported in ROCK & ROLL. These are integrated into the syntax of ROCK & ROLL by:

1. associating a reserved word with each operation name,
2. choosing a distinguished argument type for each operation, and
3. mapping each signature onto a method-invoking expression.

area_inside (*regions*)@*points*	:	*boolean*
count @*points*	:	*integer*
diameter @*points*	:	*real*
distance (*lines*)@*points*	:	*real*
equal (*points*)@*points*	:	*boolean*
intersection (*points*)@*points*	:	*points*
minus (*points*)@*points*	:	*points*
plus (*points*)@*points*	:	*points*
vertex_disjoint (*points*)@*points*	:	*boolean*
area_inside (*regions*)@*lines*	:	*boolean*
border_in_common (*lines*)@*lines*	:	*boolean*
common_border (*lines*)@*lines*	:	*lines*
equal (*lines*)@*lines*	:	*boolean*
interior @*lines*	:	*regions*
intersection (*lines*)@*lines*	:	*lines*
intersects (*lines*)@*lines*	:	*boolean*
length @*lines*	:	*real*
meets (*lines*)@*lines*	:	*boolean*
minus (*lines*)@*lines*	:	*lines*
on_border_of (*points*)@*lines*	:	*boolean*
plus (*lines*)@*lines*	:	*lines*
vertices @*lines*	:	*points*
adjacent (*regions*)@*regions*	:	*boolean*
area @*regions*	:	*real*
area_disjoint (*regions*)@*regions*	:	*boolean*
area_inside (*regions*)@*regions*	:	*boolean*
contour @*regions*	:	*lines*
edge_disjoint (*regions*)@*regions*	:	*boolean*
edge_inside (*regions*)@*regions*	:	*boolean*
encloses (*regions*)@*regions*	:	*boolean*
equal (*regions*)@*regions*	:	*boolean*
intersection (*lines*)@*regions*	:	*points*
minus (*regions*)@*regions*	:	*regions*
on_border_of (*points*)@*regions*	:	*boolean*
perimeter @*regions*	:	*real*
vertex_inside (*regions*)@*regions*	:	*boolean*

Table 1: Some Example ROSE Spatial Operations in ROCK & ROLL

Table 1 shows part of the outcome of these integration steps. The distinguished argument type follows the '`@`', and the type of the returned value follows the colon. Note that a method-invoking expression instantiates the expression to the left of the colon. Note also that many operation names are overloaded.

An example of a spatial operation in use is given in Figure 8(a). The first statement uses an embedded ROLL query to retrieve the `road` object whose `name` is `"A1"`. In the second statement a `lines` value denoted by a literal and representing an extension to the road is stored in a variable `Extension`. Finally, the compiler-generated methods `get_route` and `put_route` on `road` objects are used together with the spatial operation `plus` on `lines` to update the shape of the road with the new segments stored in `Extension`.

```
var A1        := [any R | get_name@R:Road == "A1"];
var Extension := lines{
                 [0:0_ 0:4,  2:1_ 0:4,  3:3_ 0:4],
                 [6:2_ 4:5,  6:2_ 7:5          ]};

put_route(plus(Extension)@get_route@A1)@A1;
```

(a) Extending a `road` Object

```
sum (E: Collection): points
begin
   var S := points{};
   foreach one in E do
   begin
      S := plus(get_Property()@one)@S;
   end
   S
end
```

(b) Implementing **sum** in ROCK & ROLL

Figure 8: Using Spatial Operations in ROCK & ROLL (I)

The proposed OMI in [21] also includes operations on sets of application (as opposed to spatial) objects. The operations are **sum**, **closest**, **decompose**, **overlay** and **fusion**. These are aggregation functions which may take schema-level entities (such as a property name) and may update schema-level entities (for example, by creating a new property). They also have a second-order flavour (like an *apply* operator in functional languages) insofar as they wrap around operations such as **plus**, **intersection** and **distance**. The approach proposed in [21] is slightly slanted towards a relational-like DBMS and an SQL-like query language, without being strictly dependent on either.

The context for operations on application objects in ROCK & ROLL is very different from the one assumed in [21]. It is object-oriented, strongly typed and statically type-checked. The DML is imperative, and the query language deductive, rather than SQL-like. The construction and manipulation of sets of application objects is achieved by altogether different means (as exemplified in Section 2) than those assumed in [21].

The approach to database programming that underlies ROCK & ROLL is to allow the programmer to build applications up from a lean set of built-in primitives on primary types rather than attempting to guess which of these more complex built-ins are most useful. The relational slant in [21] has led to the assumption that the DQL and the DML are one single SQL-like language, which explains the need in their proposal for built-in aggregation functions.

However, ROCK is a computationally-complete language in which not only can the operations discussed in [21] be built using the constructs in ROCK & ROLL and spatial operations like those in Table 1, but other operations that wrap around primitive spatial operations as well. For this reason, the operations that are first introduced at the OMI level of [21] are not embedded as ROCK & ROLL built-ins. ROCK & ROLL programmers can, of course, easily write code that implements equivalent operations. For example, in [21], **sum** iterates over a set of application objects $\{o_1, \ldots, o_n\}$, one of whose attributes, say a, is a spatial value v, and returns the spatial value $v' = \bigcup_{i=1}^{n} v_i$ where v_i is the spatial value assigned to a in o_i. Assuming $\{o_1, \ldots, o_n\}$ to be the components of the application object stored in the local variable `E` and a to be the property `Property: points` of each such component, a **sum** operation (specific to objects of type `Collection` and to the *points*-valued property `Property` of its component objects) can be written in ROCK & ROLL as shown in Figure 8(b). Figure 8(b) is better understood as a template specification that programmers can use to code a particular summation, i.e., while `E`, `S` and `one` are names of local variables, `Collection` and `Property` are place holders in the following sense: if a programmer instantiates `Collection` to a set- or sequence-like application type and `Property` to a *points*-valued property of the elements in the set or sequence, then the result is a piece of code that implements the aggregation function **sum** on `Property` of the component objects of `Collection`. This indicates how the aggregation functions in the OMI of [21] can be captured by existing ROCK & ROLL constructs.

Notice also that an SQL-like language does not have equivalent constructs and therefore cannot, in general, dispense with embedding in a language that does have them in the development of complete, free-standing SIS applications. Of course, the usual trade-offs arise here depending on whether a construct is made a language primitive or is left to be programmed using other, more fundamental, language primitives The former option is less flexible but opens the way for ample opportunities for optimization of storage and execution wherever the construct occurs in a program, while the latter option provides for more flexibility but makes it harder to take advantage of all the optimization opportunities presented in programs.

Declarative Methods and Queries In the extended ROCK & ROLL system, the declarative, deductive language ROLL can be used to write methods and queries which involve spatial values subject to similar constraints as the primitive types currently available in ROCK & ROLL [12].

These constraints require that ROLL methods be stratified and range-restricted, and that the evaluation of ROLL methods and queries does not effect any state transition in the database. Equivalent constraints to these are widely applied to pure, deductive database languages [10]. Their purpose is to ensure that ROLL evaluation is logically sound, order-independent and terminating. Their enforcement guarantees that ROLL methods and queries support negation and arithmetic, are capable of being comprehensively optimized using standard techniques and have a tractable [5] evaluation algorithm.

The no-state-transition constraint implies that ROLL delegates to ROCK tasks such as object creation, input/output and destructive assignment. Thus, only ROCK methods that *do not* affect variables external to it are allowed to appear in the body of a ROLL expression. The ROCK & ROLL compiler can statically check that this property holds. The no-state-transition constraint also implies that, in general, the evaluation of ROLL expressions cannot characterize an object that does not already exist in the database, since that would require a state transition to assert the existence of such object. However, in ROLL, as in Datalog, this restriction can be relaxed, e.g., for built-in arithmetic operations, if certain safeness conditions [10] hold. Similarly, when a spatial operation like those in Table 1 that yields a non-boolean value (e.g., `plus(lines)@lines: lines`) appears in a ROLL method or query, the constraint is relaxed if the same safeness criterion holds, as follows.

In ROCK and ROLL all built-in operations such as `plus(lines)@lines: lines` which construct spatial values have a functional syntax. The value returned by the operation can be regarded as an output while the recipient and arguments are considered inputs to the operation. For evaluation to be safe every input variable must have a binding resulting from the evaluation of another atom in the body of the method or the query. Note, however, that in addition, and again like arithmetic in Datalog [10] and in ROLL without spatial extensions, this binding derivation process must be stratified. Under these conditions it is safe for a new spatial value to be generated as an output from the bound input variables during evaluation.

```
var l   := lines{[0:0_0:4]};
var IRs := [ any R | get_route@R:road == Route,
                     intersects(!l)@Route];
                     on_border_of(points{0:0})@!l];
```

(a) Testing a Spatial Property in a ROLL Query

```
var l := lines{[0:0_0:4]};
var r := [ any Rest | get_route@R:road == Route,
                      minus(!l)@Route  == Rest];
```

(b) Defining Spatial Objects in a ROLL Query

Figure 9: Using Spatial Operations in ROCK & ROLL (II)

To exemplify these issues, consider Figure 9. The spatial operation appearing in Figure 9(a) yields a boolean and is safe to evaluate. The atom requesting the evaluation of a spatial operation `intersects(lines)@lines: boolean` has one ground argument (`!l`) and one variable (`Route`) that can be bound as a result of the evaluation of the other atom in the body (`get_route@R:road == Route`). The projected bindings for `R` range over the current extension of `road` in the database. Notice in the third subgoal how one can operate on more than one SDT, in this case a *points* and a *lines* value. In Figure 9(b), although a spatial value of type `lines` is generated it is still safe to do so, because the spatial operation `minus(lines)@lines: lines` returns a value that can be constructed from the ground argument (`!l`) and the binding associated with `Route` as a result of evaluating `get_route@R:road == Route`.

As an example of the functionality and usability of ROCK & ROLL consider the fire emergency application below. The program in Figure 10 uses the persistent environment `DB` comprising the types described in the paper plus others (viz. `city`, `forest`, `fire_station`) whose omission due to space restrictions should not impair comprehension too much. Virtually all language constructs have been described in the paper, one exception being the built-in `upper`, which applied to a sequence object returns the position of the last element in the sequence. The program uses a transient application-type `response`. The comments, initiated by the character '`#`', should suffice to describe the functionality displayed by the program.

This section has shown that the spatial extensions to ROCK & ROLL leave the original functionality of ROCK & ROLL intact and minimize the syntactic and semantic extensions to the available languages. A complete, free-standing SIS application in ROCK & ROLL with the spatial extensions described in this paper was presented. A self-contained example helps to demonstrate the way in which the spatially-extended ROCK & ROLL facilitates the development of SIS. Future work will be geared towards assessing the extent to which the extensions enhance the usability of the system as a platform for the development of advanced SIS applications both in terms of programmer productivity and of improved system performance.

```
program fire_emergency(DB)
begin
type response:
   properties:
      public:
         station      : name,
         at_city      : name,
         in_state     : name,
         to_fight     : regions,
         in_forests   : { name },
         alternatives : [ name ];
   ROCK:
      display();
end_type # response

class response:
   display()
   begin
      write "Station ", get_station@self, nl;
      write "in      ", get_at_city@self,
      write ", ", get_in_state@self, nl;
      write "fights fires ", get_to_fight@self, nl;
      write "in forests ", nl;
      foreach f in get_in_forests@self
      do write "   ", f, nl, nl;
      write "Alternative Stations: ", nl;
      foreach a in get_alternatives@self
      do write "   ", a, nl;
   end
end_class # response
                      # main program
 var To_Fight     : regions;
 var In_State     : name;
 var Forests      : { forest };
 var In_Forests   : { name };
 var At_Risk      : { city };
 var Suitable     : { fire_station };
 var Chosen       : int;
 var At_City      : name;
 var Alternatives := []: name;


 read To_Fight;       # These fires are raging
 read In_State;       # in this state,
                      # construct a response!
 Response := new response();
 put_to_fight(To_Fight)@Response;
 put_in_state(In_State)@Response;

                      # Which forests are affected?
 Forests := [ all Forest |
                 Fire is_in !To_Fight,
                 get_boundary@Forest:forests == B,
                 ~area_disjoint(Fire)@B ];

                      # Collect names for response.
 In_Forests := [ all Name |
                    Forest is_in !Forests,
                    get_name@Forest == Name ];
 put_in_forests(In_Forests)@Response;

                      # Which cities are at risk?
 At_Risk := [ {City} |
                  Forest is_in !Forests),
                  get_boundary@Forest == B1,
                  get_boundary@City:city == B2,
                  ~area_disjoint(B1)@B2 ];

                      # Which class A fire stations
                      # are there in cities at risk?
 Suitable := [ {FS} |
                   get_class@FS:fire_station == "A",
                   City is_in !At_Risk,
                   get_city@FS == City];

                      # Forget fire stations which
                      # lie in a road that is
                      # more than 10 km from a fire.
 foreach FS in Suitable
 do begin
       var Location := get_location@FS;
       var Routes :=
              [ {Route} |
                    get_name@State == !In_State),
                    get_network@State == Roads,
                    Road is_in Roads,
                    get_route@Road == Route,
                    vertices@Route ==  V,
                    ~vertex_disjoint(!Location)@V ];

       if   ([ | R is_in Routes,
                 distance(!To_Fight)@R < 10.0 ])
       then Alternatives ++ get_name@FS;
    end

                      # Pick one station,
 Chosen := upper@Alternatives;
 put_station(Alternatives[Chosen])@Response;

                      # find the city it is at,
 At_City := [ Name |
              get_name@FS == !Alternatives[!Chosen],
              get_city@FS == City,
              get_name@City == Name ];
 put_at_city(At_City)@Response;

                      # and leave the others as
                      # alternatives.
 Alternatives -- Alternatives[Chosen];
 put_alternatives(Alternatives)@Response;

                      # The response is now ready,
                      # display then throw away.
 display@Response;
 delete Response;

end # fire_emergency(DB)
```

Figure 10: A Fire Emergency Application in ROCK & ROLL

5 Related Work

This paper, in describing a spatial database system, can be considered to be related to a wide variety of work on geographic information systems, most of which has at least a database component. However, in most such systems the spatial data handling facilities are only coupled loosely to the database, as in ARC/INFO [27], so the focus of this section will be on spatial databases, rather than systems in which the database plays a less central role.

In recent years, there have probably been two principal approaches to the development of spatial database systems:

Kernel extensions to relational databases: Systems in this category extend the relational model and an associated query language with spatial data handling facilities [6, 13, 18]. As relational database languages are not powerful enough to be used to implement core spatial data types, spatial extensions are generally implemented by extending the kernel of the database or by exploiting abstract data type facilities. The advantages of these systems are that they provide declarative querying and special purpose storage management for spatial information. The principal disadvantage is the lack of integrated programming facilities and the limited modelling power of the relational model for the aspatial aspects of a domain.

ROCK & ROLL improves upon the capabilities of these systems by providing a fully-fledged structural and behavioural object-oriented model, by implementing a modern approach to database programming using persistent programming language technology, and by making available to application designers the full power of safe, tractable, declarative rule-based programming for spatial inference.

Layered extensions to object-oriented databases: Systems in this category exploit the enhanced programming facilities supported by most commercial object-oriented databases to develop a library of spatial types as a layer on top of the database [30, 31]. Such systems generally provide powerful programming environments for spatial applications, but the lack of kernel support for implementing spatial primitives is likely to cause performance problems, and few such systems support comprehensive declarative querying facilities.

ROCK & ROLL improves on such systems by providing a better integration of spatial types into the database programming environment, specialized storage management and spatial optimization techniques built into the system and the basic mechanisms for making use of methods relying on spatial inference.

The aim of the project from which this paper emanates is to combine the benefits of both of the above approaches, by providing kernel support for spatial types in an environment that supports both declarative querying and imperative programming. Also related to the work presented here is that which seeks to exploit deduction within spatial systems, such as to handle multiple representations [25], to support general derived properties [14], or to assist with automatic feature extraction [3]. The spatial extensions to ROCK & ROLL are complementary to the above, in that ROCK & ROLL can be seen as a powerful implementation platform for supporting deduction in spatial databases.

The potential of deductive object-oriented databases for spatial data handling has been recognized by other researchers [23, 26], but we are not aware of other systems that provide kernel support for spatial types in conjunction with strongly typed imperative and declarative language facilities and an object-oriented data model, as described here.

6 Summary of Benchmarks

The benchmark [11, 22, 29] applied to the spatially-extended ROCK & ROLL DOOD system concentrated on obtaining insights into the effectiveness of the spatial techniques selected for use, the benefits derived from moving spatial data handling capabilities into the kernel of the system and the effectiveness of a spatial DOOD for the development of complex spatial data handling applications. It is also intended to apply benchmark techniques to ascertain the impact of logic-based query optimization techniques and the comparative merit of bottom-up and top-down [26] evaluation strategies, but these have not been investigated yet.

The benchmark process was developed in-house [11] thus allowing a fine-grained feedback on the performance of specific operations. This not only facilitated comparison with other systems but helped identify weaknesses in the implementation which could be addressed in the lifetime of the research project within which the results reported in this paper were obtained.

It is clearly necessary for performance analyses to be carried out using real world data, rather than that generated for performance assessment (especially using an in-house benchmark). It is envisaged that this task will be carried out in close cooperation with Ordnance Survey, which are industrial partners in the research project whose preliminary results this paper reports.

We will refer to ROCK & ROLL with spatial data handling facilities inside the kernel as S-R&R and outside (i.e., specified using the general purpose facilities of the ROCK and ROLL languages) as **R&R**.

The benchmark was first applied to **R&R** and **S-R&R** to see the effect of moving functionality into the kernel. Subsequently, the benchmark was applied to **S-R&R** and Postgres95 [33] to see how **S-R&R** performs in comparison with an advanced, general-purpose object-relational DBMS.

For reasons of space, we must refer the reader to [11, 22, 29] for a full description of the benchmark used, as well as a survey and comparison with two other spatial benchmarks [32, 17], and the conditions and results of applying the benchmark to ARC/INFO. For the conditions and the application of the benchmark to Postgres95, see [29]. For the comparison between the performance of the extended and the non-extended versions of ROCK & ROLL as well as a fuller analysis of the most recent results, see [22]. [4]

In the remainder of this section, the most important conclusions drawn in [11, 22, 29] are summarized.

S-R&R v. R&R

- **S-R&R** is much faster than **R&R** for all queries, on single or multiple runs (cf. [22]).
- As regards scalability, as reflected by the timings for the small, medium or large datasets which the benchmark supplies, both **S-R&R** and **R&R** seem to scale up roughly linearly or better from small to medium and medium to large, with only one case of worse than linear.
- Times for **S-R&R** degrade less with increase in data size than the corresponding times for **R&R** (cf. [22]). This effect is likely to be due to the use of superior data structures and algorithms for spatial operations.

S-R&R v. Postgres95

- With respect to absolute timings, for most queries they fall within the same order of magnitude. For three queries, **S-R&R** performed worse than Postgres95, but achieved similar or better results for the other twenty-six queries (cf. [22]).
- There were indications that Postgres95, like **R&R**, may impose a very severe performance penalty if poorer data structures and algorithms for spatial operations are used, as they seem to be, than those embedded in the kernel of **S-R&R**.

In conclusion, the benchmarks have shown that there is a performance advantage in embedding spatial operations in the kernel as opposed to programming them using general purpose facilities. These were sufficiently significant to allow **S-R&R** to achieve comparable performance to a system like Postgres95, even though, unlike Postgres95, **S-R&R** has no control over the physical storage layer, depending as it does on a third-party object manager.

7 Conclusions

This paper has described the embedding of a ROSE algebra of spatial values into the ROCK & ROLL DOOD system. The contributions of the work described in this paper are the following:

- A treatment of SDTs as embedded, primitive types of a host DBMS, thereby allowing developers of spatial-data handling applications to conceive of spatial values in the same way as they conceive of, e.g., boolean values, number values and string values.
- The smooth integration of a ROSE-based spatial data handling component with the ROCK & ROLL DOOD system, thereby yielding a platform for SISs that overcomes the limitations of current, mainstream database technology and at the same time ensuring a consistent treatment of geometries and the kind of computational efficiency made possible by the presence of built-in spatial object manager and query optimization.
- A platform in which DOOD technology is made available within a domain, viz., SISs, in which both deductive inference and object modelling are recognized as crucial for the smooth development of the kind of complex problems that make SISs as challenging an application area as it is relevant to the public at large.

As a result, the extended ROCK & ROLL DOOD system is able to deliver the following advantages over similar attempts to integrate spatial data handling functionalities to advanced DBMSs:

[4] The documents cited as [11, 22, 29] are available, up on request, from the fourth author, e.g., via email at `howard@cee.hw.ac.uk`.

- the first persistent, DOOD programming environment with integrated spatial data handling functionality built into the kernel, in contrast to approaches relying on a library of SDTs (e.g., our own previous work [1, 2]);
- measurements taken from the current prototype of the extended system, which is still under construction, indicate improvements in performance of two orders of magnitude in average over the layered implementation described in [1]. These improvements stem from:
  - the computational efficiency of the spatial operations of the ROSE algebra [19],
  - a novel approach to the low-level implementation of realms [28],
  - a query optimizer that is aware of the semantics of spatial types and seamlessly integrates spatial and aspatial query optimization,
  - the support for spatial object management and indexing in the DBMS kernel.

The extended ROCK & ROLL system provides much more comprehensive and better integrated database programming facilities than other candidate platforms for spatial information systems. The extended system provides developers with an intuitive, expressive, formally defined collection of spatial data types as primitive types whose operations have state-of-the-art computational complexity. The integration of these types with the object-oriented modelling, imperative programming and deductive querying facilities of ROCK & ROLL makes available a comprehensive and integrated suite of complementary mechanisms for the development of spatial information systems.

Acknowledgements The work that resulted in this paper is part of the **ODESSA: Object-oriented Deductive Spatial Systems Architecture** project funded by the Engineering and Physical Sciences Research Council through the AIKMS programme, and their support is gratefully acknowledged. We would like to thank Mr Neil Smith and Mr Glen Hart of Ordnance Survey as the industrial partners in the ODESSA project.

We also thank Volker Müller for many useful discussions on the subjects addressed in this paper and for his implementation of the *realm* infrastructure on which the ROCK & ROLL extensions described in this paper depend. Finally we thank Alan Patrick, Kosmas Dietrich and Reto O. Hafner for their work in benchmarking the spatial extensions to ROCK & ROLL.

The support from the ERASMUS programme sponsored by the European Commission which allowed Volker Müller, Kosmas Dietrich and Reto O. Hafner to complete their degrees at Heriot-Watt University for an academic year is gratefully acknowledged, as is the support to Alan Patrick provided by the UK EPSRC.

References

[1] Alia I. Abdelmoty, Norman W. Paton, M. Howard Williams, Alvaro A.A. Fernandes, Maria L. Barja, and Andrew Dinn. Geographic Data Handling in a Deductive Object-Oriented Database. In *[24]*, pages 445–454, 1994.

[2] Alia I. Abdelmoty, M. Howard Williams, and Norman W. Paton. Deduction and Deductive Databases for Geographic Data Handling. In *[4]*, pages 443–464, 1993.

[3] Alia I. Abdelmoty, M. Howard Williams, and Michael Quinn. A Rule-Based Approach to Computerized Map Reading. *Information and Software Technology*, 35(10):587–602, October 1993.

[4] David Abel and Beng Chin Ooi, editors. *Advances in Spatial Databases – 3rd International Symposium SSD'93*, number 692 in LNCS, Singapore, June 1993. Springer-Verlag. ISBN 3-540-56869-7.

[5] Serge Abiteboul, Richard Hull, and Victor Vianu. *Foundations of Databases.* Addison-Wesley Publishing Company, Reading, MA, USA., 1995.

[6] W.G. Aref and H. Samet. Optimisation Strategies for Spatial Query Processing. In G. Lohman, A. Sernadas, and R. Camps, editors, *Proc. 17th Int. Conf. on Very Large Data Bases (VLDB)*, pages 81–90. Morgan-Kaufmann, 1991.

[7] Maria L. Barja, Alvaro A.A. Fernandes, Norman W. Paton, M. Howard Williams, Andrew Dinn, and Alia I. Abdelmoty. Design and Implementation of ROCK & ROLL: A Deductive Object-Oriented Database System. *Information Systems*, 20(3):185–211, 1995.

[8] Maria L. Barja, Norman W. Paton, Alvaro A.A. Fernandes, M. Howard Williams, and Andrew Dinn. An Effective Deductive Object-Oriented Database Through Language Integration. In Jorge Bocca, Matthias Jarke, and Carlo Zaniolo, editors, *Proceedings of the Twentieth International Conference on Very Large Data Bases*, pages 463–474, Santiago, Chile, September 1994. Morgan Kaufmann Publishers, Inc.

[9] Peter Buneman and Sushil Jajodia, editors. *Proceedings of the 1993 ACM SIGMOD International Conference on the Management of Data*, Washington, DC, USA., May 1993. ACM Press. SIGMOD Record, 22(2), June 1993.

[10] Stefano Ceri, Georg Gottlob, and Letizia Tanca. *Logic Programming and Databases.* Surveys in Computer Science. Springer-Verlag, Berlin, Germany., 1990. ISBN 3-540-51728-6, 0-387-51728-6.

[11] Kosmas Dietrich. Benchmarking Spatial Databases, 1995. Department of Computing and Electrical Engineering, Heriot-Watt University, Edinburgh, UK. Final Year Honours Dissertation.

[12] Andrew Dinn, Norman W. Paton, M. Howard Williams, Alvaro A.A. Fernandes, and Maria L. Barja. The Implementation of a Deductive Query Language Over an Object-Oriented Database. In T. W. Ling, A. O. Mendelzon, and L. Vieille, editors, *Proceedings DOOD'95 – Fourth International Conference on Deductive and Object-Oriented Databases*, LNCS 1013, pages 143–160, Singapore, December 1995. Springer-Verlag. ISBN 3-540-60608-4.

[13] Max J. Egenhofer. Spatial SQL: A query and presentation language. *IEEE Trans. Knowledge and Data Eng.*, 6:86–95, 1994.

[14] Max J. Egenhofer and Andrew U. Frank. LOBSTER: Combining AI and Database Techniques for GIS. *Photogrammetric Engineering & Remote Sensing*, 56(6):919–926, 1990.

[15] Max J. Egenhofer and John R. Herring, editors. *Advances in Spatial Databases: 4th International Symposium, SSD'95*, number 951 in LNCS, Portland, USA, August 1995. Springer-Verlag. ISBN 3-540-60159-7.

[16] Alvaro A.A. Fernandes, Maria L. Barja, Norman W. Paton, and M. Howard Williams. The Formalization of ROCK & ROLL: A Deductive Object-Oriented Database System. *Information and Software Technology*, 39(6):379–389, 1997.

[17] Béatrice Finance. The GIS Benchmark Specification, 1993. PYTHAGORAS Esprit III Project Deliverable.

[18] Ralf Hartmut Güting. Gral: An Extensible Relational Database System for Geometric Applications. In *Proceedings of the Fifteenth International Conference on Very Large Data Bases*, pages 33–44, 1989.

[19] Ralf Hartmut Güting, Thomas de Ridder, and Markus Schneider. Implementation of the ROSE Algebra: Efficient Algorithms for Realm-Based Spatial Data Types. In *[15]*, pages 216–239, 1995.

[20] Ralf Hartmut Güting and Markus Schneider. Realms: A Foundation for Spatial Data Types in Database Systems. In *[4]*, pages 14–35. 1993.

[21] Ralf Hartmut Güting and Markus Schneider. Realm-Based Spatial Data Types: The ROSE Algebra. *VLDB Journal*, 4:243–286, 1995.

[22] Reto O. Hafner. Benchmarking of ROCK & ROLL: A Deductive Object-Oriented Database System, 1996. Department of Computing and Electrical Engineering, Heriot-Watt University, Edinburgh, UK. Final Year Honours Dissertation.

[23] Christopher B. Jones and L.Q. Luo. Hierarchies and Objects in a Deductive Spatial Database. In *Proc. SDH*, pages 588–603. Taylor & Francis, 1994.

[24] Dimitri Karagiannis, editor. *Database and Expert System Applications - 5th International Conference, DEXA '94, Proceedings*, LNCS 856, Athens, Greece., September 1994. Springer-Verlag. ISBN 3-540-58435-8.

[25] David B. Kidner and Christopher B. Jones. A Deductive Object-Oriented GIS for Handling Multiple Representations. In T.C. Waugh and R.G. Healey, editors, *Proc. 6th Int. Symp. on Spatial Data Handling (SDH 94)*, pages 882–900. AGI/IGU Commission on GIS, 1994.

[26] W. Lu and J. Han. Query Evaluation and Optimization in Deductive and Object-Oriented Spatial Databases. *Information and Software Technology*, 37(3), 1995.

[27] S. Morehouse. ARC/INFO: A Geo-Relational Model for Spatial Information. In *Proceedings of 7th Int. Symposium on Computer Assisted Cartography*, pages 388–398, Washington, DC, 1986.

[28] Volker Müller, Norman W. Paton, Alvaro A.A. Fernandes, Andrew Dinn, and M. Howard Williams. Virtual Realms: An Efficient Implementation Strategy for Finite Resolution Spatial Data Types. In M.J. Kraak and M. Molenaar, editors, *Advances in GIS Research II, Proceedings Vol. 2., 7th International Symposium on Spatial Data Handling - SDH'96*, pages 11B.1–11B.13, Delft, The Netherlands, August 1996. IGI.

[29] Alan Patrick. A Comparison of Two Extensible Database Systems for the Development of a Spatial Database. Master's thesis, Department of Computing and Electrical Engineering, Heriot-Watt University., Edinburgh, UK., 1995.

[30] S.A. Roberts and M.N. Gahegan. An Object-Oriented Geographic Information System Shell. *Information and Software Technology*, 35(10), 1993.

[31] M. Scholl and A. Voisard. Geographic Applications: An Experience with O_2. In *Building an Object-Oriented Database System: The story of O_2*, pages 585–618. Morgan-Kaufmann, 1992.

[32] Michael Stonebraker, James Frew, Kenn Gardels, and Jeff Meredit. The Sequoia 2000 Benchmark. In Peter Buneman and Sushil Jajodia, editors, *[9]*, pages 2–11. 1993.

[33] A. Yu and J. Chen. *The Postgres95 User Manual (Beta Release 0.02)*, 1995. Available with the software release.